Włodzimierz BIELECKI, **Krzysztof SIEDLECKI**, Sławomir WERNIKOWSKI
WEST POMERANIAN UNIVERSITY OF TECHNOLOGY, DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY,
ul. Żołnierska 49, 71-210 Szczecin

An approach to form affine time partitioning for statement instances of arbitrarily nested loops

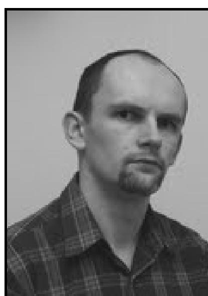
Prof. Wlodzimierz BIELECKI

He is a head of the Software Technology Department of the West Pomeranian University of Technology, Szczecin. His research interest includes parallel and distributed computing, optimizing compilers, extracting both fine- and coarse grained parallelism available in program loops.

e-mail: wbielecki@wi.zut.edu.pl

Ph.D. eng. Krzysztof SIEDLECKI

He has graduated and obtained his Ph.D. degree in Computer Science from the Technical University of Szczecin, Poland. His research focus on optimizing compilers, parallel and distributed processing, software engineering.



e-mail: ksiedlecki@wi.zut.edu.pl

M.Sc. eng. Sławomir WERNIKOWSKI

Senior lecturer at Software Technology Department of the West Pomeranian University of Technology, Szczecin. His research interests are focused on programming techniques and paradigms as well as multiprocessor GPUs utilization in general-purpose computing.

e-mail: swernikowski@wi.zut.edu.pl

Abstract

A novel approach to form affine time partitioning for statement instances of arbitrary nested loops is presented. It is based on extracting free-scheduling which next is used to form a system of equations to produce legal time partitioning. The approach requires an exact dependence analysis. To carry out experiments, the dependence analysis by Pugh and Wonnacott was chosen. Examples illustrating the approach and the results of experiments are presented.

Keywords: loop parallelization, free schedule, affine time partitioning.

Wyznaczenie harmonogramu instancji instrukcji dla pętli dowolnie zagnieżdżonych

Streszczenie

Przedstawiona została nowa metoda do tworzenia afinicznych odwzorowań czasowych instancji instrukcji dla pętli dowolnie zagnieżdżonych. Metoda bazuje na ekstrakcji harmonogramu swobodnego, wykorzystywanego do tworzenia legalnego odwzorowania czasowego. Metoda wymaga dokładnej analizy zależności. Do przeprowadzenia eksperymentów, wybrana została analiza zależności zaproponowana przez Pugh'a and Wonnacott'a. W analizie tej zależności reprezentowane są przez relacje zależności, natomiast przestrzeń iteracji przez zbiory. Do tworzenie zbiorów i relacji zależności wykorzystywana jest arytmetyka Presburgera. Zostały przedstawione przykłady ilustrujące działanie metody dla pętli idealnie zagnieżdżonej, jak i dla pętli nieidealnie zagnieżdżonej. Eksperymenty przeprowadzone zostały na procesorach graficznych firmy nVidia z wykorzystaniem technologii CUDA w trybie zgodności z wersją 1.1. Wyniki zostały przedstawione w formie tabelarycznej. Zostały przedstawione prace pokrewne oraz kierunek dalszych badań.

Słowa kluczowe: zrównoleglenie pętli, harmonogram swobodny, afiniczne odwzorowanie czasowe.

1. Introduction

The Affine Transformation Framework (ATF) unifies a large class of transformations, including loop interchange, reversal, skewing, fusion, fission, reindexing, scaling and statement reordering [7, 8]. In this framework, instances of each loop statement are identified by the loop index values of their surrounding loops, and affine expressions are used to map these loop index values into:

- *Space partitions*, where loop statement instances belonging to the same space partition are mapped to the same processor;
- *Time partitions*, where loop statement instances belonging to time partition *i* execute before those in partition *i*+1. This execution order can be given by a loop whose *i*-th iteration executes all statement instances belonging to partition *i*.

In this paper, we concentrate only on time partitioning and propose a novel technique permitting for building affine time partitioning characterized by reduced computing complexity in comparison with well-known techniques.

2. Background

In this paper, we deal with affine loop nests where lower and upper bounds as well as array subscripts and conditionals are affine functions of surrounding loop indices and possibly of structure parameters, and the loop steps are known constants.

A loop is perfectly nested if all its statements are comprised within the innermost nest. Otherwise, the loop is called arbitrarily nested.

Following work [2], we refer to a particular execution of a statement for a certain iteration of the loops, which surround this statement, as an operation.

Two operations *J* and *I* are dependent if both access the same memory location and if at least one access is a write. We refer to *I* and *J* as the source and destination of a dependence, respectively, provided that *I* accesses the same memory location earlier than *J* ($I \prec J$).

To describe the algorithm and carry out experiences, we chose the dependence analysis proposed by Pugh and Wonnacott [3] where dependences are represented with dependence relations of the form

$$\{[input\ list]\text{->}[output\ list] : constraints\}, \quad (1)$$

where *input list* and *output list* are the lists of variables and/or expressions used to describe input and output tuples, and *constraints* is a Presburger formula describing the constraints imposed upon *input list* and *output list*.

A dependence relation is a mapping from one iteration space to another, and is represented by a set of linear constraints on variables that stand for the values of the loop indices at the source and destination of a dependence, and the values of the symbolic constants.

A one-dimensional affine partition mapping for a statement *s* in a loop is an affine expression:

$$F_s = C_s * I_s + c_s, \quad (2)$$

where C_s is an integer $1 \times n_s$ matrix, and c_s is an integer, n_s is the number of loops surrounding statement s. Consider a p-statement loop originating dependences represented with a set of n dependence relations

$$R_k = \{ s1_k(I_k) \text{ -> } s2_k(J_k) : constraints_k \}, \quad (3)$$

where k = 1, 2,…,n and $s1_k$, $s2_k$ are the identifiers of statements whose instances originate sources and destinations of dependences described with R_k, respectively ($s1_k$, $s2_k \in$ [1,2,…, p]). In case of time partitioning for each statement s, we have to find an affine mapping (2) such that

$$F_s1_k (I_k) \le F_s2_k (J_k), \quad (4)$$

guaranteeing that in the transformed loop, the destination of a dependence will be executed no earlier than the corresponding dependence source. We can rewrite condition (4) as follows

$$C_{s2k} *J_k + c_{s2k} - C_{s1k} * I_k - c_{1k} \ge 0, \quad (5)$$

where k = 1, 2,…, n, C_{s1k} and C_{s2k} are $1 \times n_{s1}$, $1 \times n_{s2}$ matrices, respectively, while c_{s1k} and c_{s2k} are integers representing constant terms of affine mappings.

Time partitioning is known also as scheduling [7].

Definition 1 [4]

A free schedule assigns operations as soon as their operands are available, that is, mapping $\sigma: I \to Z$ such that

$$\sigma(p) = \begin{cases} 0 \ \textit{if there is no } p' \in I \ s.t. \ p' \prec p \\ 1 + \max(\sigma(p'), p' \in I, p' \prec p) \end{cases} . \quad (6)$$

The free schedule is the "fastest" schedule possible. Its total execution time is:

$$T_{free} = 1 + \max(q_{free}(p), p \in I) \quad (7)$$

Below, we present the main idea of the algorithm, extracting the free schedule for a given loop, details of this algorithm are presented in [1]. Following that algorithm, for each statement all operations are divided into two sets containing independent and dependent operations (sources and destinations of dependences), respectively. Using the second set, those operations are found for which all operands are available. They form the operations of layer Lay_1 to be executed firstly (the first time partition). The operations of Lay_1 are eliminated from the second set, and the algorithm finds again those operations for which all operands are available, forming the operations of the second time partition. This process is repeated until there are no operations in the second set. The operations of the first set can be combined with the operations of arbitrary levels. Operations in each time partition can be executed in parallel.

In arbitrarily nested loops, statements may have different domains. In general, to find free schedule, we should deal with each statement independently as presented in paper [1].

Using sets Lay_1, Lay_2, …, we can produce parallel code using any well-known technique, for example, the function codegen in the Omega library [5].

Let us illustrate the algorithm by means of the following example.

Example 1:

```
for i=1 to 6 by 1 do
  for j=1 to 10 by 1 do
    a(i+j,3*i+j+3)=a(i+j+1,i+2*j+4)
```

Using Petit [5], we extract the following dependence relations:

```
R1 = {[i,2i]->[i,2i+1]: 1 <= i <= 4},
R2 = {[i,j]->[i',i+i'+1]: j=2i' && 1<=i<i'<= 5},
R3 = {[i,j]->[j-i-1,2i]: 2i+2<=j<=i+7,10 && 1<=i}.
```

Following the algorithm presented in [1] to extract the free schedule for this example, we get three layers Lay_1, Lay_2, Lay_3 where operations in each layer can be executed in parallel:

```
Lay1 = {[1,2]} union {[i,j]: Exists( alpha :
    2alpha = j && 2i+2 <= j <= 10 && 1 <= i)}
    union {[i,j]: 2i+2 <= j <= i+7, 10 && 1 <=
    i},
Lay2 = {[1,3]} union {[i,In_2]: i <= In_2-2, 5 &&
    In_2 <= 2i} union {[i,In_2]: Exists( alpha :
    2alpha = In_2 && 2 <= In_2 <= 2i-2, -2i+18 &&
    i <= 6)},
Lay3 = {[i,2i+1]: 2 <= i <= 4}.
```

There must be explicit synchronization between the execution of layers (time partitions). Figure 1 presents the iteration space with dependences and the layers for the examined example.

There are also independent operations contained in set IND:

```
IND = {[1,3]} union {[1,1]} union {[i,j]: 2,j-4 <=
      i <= 6 && 1 <= j <= 2i-1} union {[5,10]}
      union {[1,9]},
```

that can be combined with operations of arbitrary layers.

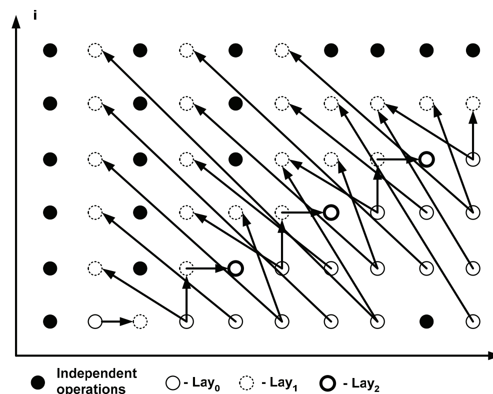


Fig. 1. Dependences and Layers for Example 1
Rys. 1. Zależności i poziomy dla przykładu 1

3. Algorithm to form time partitions

The idea of the algorithm presented in this section is as follows. We extract firstly sets Lay_{1i} , i=1,2,…, p, following the algorithm presented in [1]. Then, we generate equations to find unknowns representing affine time partitioning. From all solutions to those equations, we choose minimal non-zero solution such that it preserves all loop dependences, i.e., this solution forms legal affine time mapping. If there is no such a solution, we form sets Lay_{2i} i=1,2,…, p, and repeat the above process until yielding an appropriative affine time mapping.

Algorithm 1. Finding time partitioning for loop statement instances

Input: 1) Set S containing relations R_i, i=1,2,…, n where n is the number of relations, describing all the dependences in a loop; 2) the number of loop statements, p; 3) the maximal number of the algorithm iterations, N.

Output: Matrices C_{ij} of size $1 \times n_i$ and integers c_i presenting affine time partitioning, i=1,2,…,p; j=1,2,…,n_i where n_i is the number of loops surrounding statement i.

Method:

1. $k = 1$
2. Extract sets Lay_{ki} , i=1,2,…, p, using set S and the algorithm presented in [1].
3. i=1

   3.1 While Lay_{ki} is empty AND $i \neq p$ do i=i+1;
   $e = i$, form the following system of equations

$$\sum_{s=1}^{n_s} C_{es} X_e = \sum_{s=1}^{n_s} C_{es} Y_e \;\&\&$$
$$\text{exists } X_e,\, Y_e \text{ s.t. } X_e, Y_e \in Lay_{ki} \;\&\&\; X_e \prec Y_e \qquad (8)$$

   /* note that the system above is to find some elements X_e Y_e belonging to Lay_{ke} but not all the elements belonging to the same time partition under the free schedule contained into Lay_{ke} */

   3.2 if $i \neq p$ then i= i+1;
   while Lay_{ki} is empty AND $i \neq p$ do i=i+1;
   $e' = i$; if $e' \neq e$, then form the following system of equations

$$\sum_{s=1}^{n_s} C_{es} X_e + c_e = \sum_{s=1}^{n_{s'}} C_{e's} Z_{e'} + c_{e'} \;\&\&$$
$$\text{exists } X_e,\, Z_e \text{ s.t. } X_e \in Lay_{ke} \;\&\&\; Z_e \in Lay_{ke} \qquad (9)$$

   /* note that the system above is to find some elements X_e Z_e belonging to Lay_{ke} and $Lay_{ke'}$, respectively, but not all the elements belonging to the same time partition under the free schedule contained into Lay_{ke} and $Lay_{ke'}$ */

   3.3 if $i \neq p$ then go to step 3.2
4. Resolve each system formed in step 3 for C_{es} , $C_{e's}$, X_e, Y, $Z_{e'}$.
5. From all the solutions to the systems formed in step 3, for each loop statement, choose a minimal (with minimal values) legal non-zero solution (if appropriate) C_{s1k}, C_{s2k}, c_{s1k}, and c_{s2k} such that for each relation R_k , k = 1, 2,…, n in set S the following condition is satisfied

$$C_{s2k} J_k + c_{s2k} - C_{s1k} I_k - c_{s1k} > 0 \;\&\&$$
$$J_k, I_k \text{ satisfy constraints of relation } R_k \qquad (10)$$

6. If there does not exist any legal solution, then
   if k<N, then k = k+1 and go to step 2; otherwise the end, the algorithm fails to extract affine time partitions.

Let us illustrate the presented algorithm by means of the following example.

Example 2:

```
for i=1 to 10 do
  for j=1 to 10 do
1:  a(i,j+3) = a(i+1,2*j+1);
```

Using Petit[5,6], we extract the following dependence relation

```
R1 = {[i,j]->[i+1,2j-2] : 1<=i<= 9 && 2<=j<= 6}.
```

Following the algorithm [1], we extract the following set

```
Lay11 = {[1,j]: 2 <= j <= 6} union
        {[i,j]: Exists(alpha : 2alpha = 1+j && 2
        <= i <= 9 && 3 <= j <= 5)}
```

To find affine time partitioning, we form the following equation

```
C11*x1 + C12*x2 = C11*y1 + C12*y2
```

that correspondents to the system (8). The constraint *exists* X_e, Y_e s.t. $X_e, Y_e \in Lay_{ki}$ for this example is as follows

```
((x1=1 && x2=2) || (x1=1 && x2=3) || ...) &&
((y1=1 && y2=2) || (y1=1 && y2=3) || ...).
```

The constraint $X_e \prec Y_e$ is of the form

```
x1 < y1 || (x1=y1 && x2<y2).
```

Putting all together, we form the following system of equations

```
C11*x1 + C12*x2 = C11*y1 + C12*y2 &&
((x1=1 && x2=2) || (x1=1 && x2=3) || ...) &&
((y1=1 && y2=2) || (y1=1 && y2=3) || ...) &&
(x1<y1 || (x1=y1 && x2<y2))
```

Using Mathematica [9], we get the following solution to the system above:

```
C11 = 0 && C12 = Integers
C11 = Integers && C12 = 0
```

We choose the solution C_{11} = 1, C_{12} = 0 and following step 5 check whether this solution is legal

```
C11*(i+1) + C12*(2j-2) - C11*i - C12*j = C11 +
C12*(j-2) = C11 = 1
```

for all `1 <= i <= 9 && 2 <= j <= 6}`

Hence, the chosen solution is a valid affine time partitioning. Let us consider another loop.

Example 3:

```
for i=1 to 5 do
 1: a(i,i) = c(i,i)
    for j=1 to 5 do
 2:    b(i,j)=a(i,j+1)
```

Petit exposes the following dependence relation for this loop

```
R1 = {[i] -> [i,i-1] : 2 <= i <= 5}.
```

The algorithm presented in [1] produces the following sets.

```
Lay11= {[i]: 2<=i<=5}, Lay12={[i,i-1]: 2<=i<= 5}.
```

Following step 3.1 of the presented algorithm, we get

```
C11*x1 = C11*y1 && (x1=2 || x1=3 || x1=4 || x1=5) &&
(y1=2 || y1=3 || y1=4 || y1=5) && x1<y1).
```

While step 3.2 yields the following system

```
(C11*x1 + c1 = C21*z1 + C22* z2 + c2 && (x1=2 || x1=3
|| x1=4 || x1=5) && ((z1=2 && z2=1) || (z1=3 &&
z2=2) || (z1=4 && z2=3) || (z1=5 && z2=4))).
```

Applying Mathematica [9], we get a set of solutions to the system above and next examine each solution to choose a legal solution. Such a solution is of the form

```
z2=1 && z1=2 && x1= 3 && c2=c1+3C11 - 2C21 - C22.
```

From this solution, we get

```
C11=1, c1=0, C21=1, C22=0, c2=1.
```

This solution (for `2 <= i <= 5`) is legal, because

```
C21*i + C22(i-1) +c2 - C11*i -c1 = 1*i+0+1-1*i-0 = 1
```

In Table 1, we summarize the results for the examined examples. The second column contains the number of layers needed to find a legal solution. The last column presents affine time partitioning.

Tab. 1. Summary of examined examples
Tab. 1. Podsumowanie przykładów testowych

Example	Layers to find solution	Partitioning
1	1	C_{11}=1, C_{12}=0
2	1	C_{11}=1, c_1=0, C_{12}=1, C_{22}=0, c_2=1

4. Results of experiments

The presented algorithm was applied to the following loop:

```
for i=1 to N do
  for j=1 to N do
1:  a[i , j] = hypotf(a[i-1 , j]/sqrtf(2.),
                      a[i-1 , j]/sqrtf(2.));
```

to get the following affine transformation C_{11}=1, C_{12}=0 that permits for producing the following parallel loop

```
seqfor i=1 to N do
  parfor j=1 to N do
1:  a[i , j] = hypotf(a[i-1 , j]/sqrtf(2.),
                      a[i-1 , j]/sqrtf(2.));
```

where `seqfor` and `parfor` denote the sequential and parallel loop, respectively (details on how to generate code on the basis of affine transformations see in [4]).

To demonstrate that the fine-grained parallelism represented by the loop above is appropriate for the graphic processor units (GPUs) [12], we carried out experiments on the following computer:

- 2 Intel Quad-Core Xeon processors (E5440 model), running at 2.83GHz, each equipped with 6 MB L2 internal cache (8 cores in total);
- 8 GB physical RAM;
- graphic adapter manufactured by XFX Pine Group Inc, equipped with nVidia GTS 250 GPU (CUDA capability level 1.1) running at 1.836 GHz, utilizing 511 MB onboard global memory space;
- Debian GNU/Linux 5.0 (AMD64 architecture) running kernel 2.6.26;
- nVidia kernel driver module version 3.0 (195.36.15) for Linux x86_64;

The results of experiments are presented in Tables 2 where the first column represents the number of the loop iterations, the second column shows the array sizes, and the next columns contain the time of the sequential and parallel programs execution in the CUDA GPUs.

The execution time was measured using the *clock_gettime()* function called in the *CLOCK_PROCESS_CPUTIME_ID* mode. Total execution time is the sum of delivering data to device memory (*deliver*), actual loop execution (*process*) time, and fetching results from device memory (*fetch*).

Tab. 2. Results of experiments
Tab. 2. Wyniki badań eksperymentalnych

N	Array size, MB	GPU sequential execution time, ms			
		deliver	process	Fetach	total
1024	4	2,1	20,3	2,8	25,2
2048	16	8,6	54,2	10,1	72,9
4096	64	33,8	163,2	39,1	236,1
8192	256	134,9	543,9	154,7	833,5
10240	400	210,0	816,5	240,5	1267,0

Tab. 3. Results of experiments
Tab. 3. Wyniki badań eksperymentalnych

N	Array size, MB	Execution time, MS					t_{cpu}/t_{gpu}
		t_{cpu}	GPU paralel				
			deliver	process	fetch	total (t_{gpu})	
1024	4	30,5	2,0	2,0	2,7	6,7	4,6
2048	16	150,4	8,3	4,7	10,3	23,3	6,5
4096	64	606,9	33,8	14,6	39,4	87,8	6,9
8192	256	2441,0	134,9	37,5	155,4	327,8	7,4
10240	400	3686,3	209,1	62,1	239,6	510,8	7,2

The results of experiments demonstrate that applying affine time partitioning allows for using multiple GPUs to reduce the execution time of parallel programs.

5. Related work

Well-known techniques to form affine time partitioning are based on forming system of equations (4) – see background – and then such a system should be resolved applying standard linear algebra techniques (Gaussian Elimination, Fourier-Motzkin Elimination) as well as the Affine Form of Farkas Lemma [2, 4, 8]. Affine mappings are represented by all linearly independent solutions [C_s1_k c_s1_k] and [C_s2_k c_s2_k] to system (5) and valid for any I_k, J_k satisfying constraints k , k = 1, 2, ..., n. These linearly independent solutions can be found by means of techniques presented in [10, 11].

The main drawback of well-known techniques is the considerable computing complexity that prevents their usage in optimizing compilers.

The technique, presented in this paper, is based on forming a limited number of linear equations that can be resolved by means of standard algorithms whose complexity is much less than that of classic techniques permitting for extracting affine time partitioning.

6. Conclusion and future work

In this paper, we presented a technique permitting for building affine time mapping. In comparison with well known-techniques, it has reduced computing complexity and can be applied to parallelized arbitrarily nested loops. In our future work, we plan to implement this approach and verify it on real-life codes to be parallelized and run in GPUs.

The work was supported by finances of West Pomeranian Provincial Administration.

7. References

[1] Bielecki W., Siedlecki K.: Finding free schedules for non-uniform loops. Euro-Par 2003, 26-29.08, Klagenfurt, Austria, LNCS, vol. 2790, pp. 297-302.
[2] Vivien F.: On the optimality of Feautrier's scheduling algorithm. In Proceedings of the EUROPAR'2002, 2002.
[3] Pugh W., Wonnacott D.: An Exact Method for Analysis of Value-based Array Data Dependences. Workshop on Languages and Compilers for Parallel Computing, 1993.
[4] Darte A., Robert Y., Vivien F.: Scheduling and Automatic Parallelization. Birkhäuser Boston, 2000.
[5] The Omega project. http://www.cs.umd.edu/projects/omega
[6] Pugh W., Wonnacott D.: An Exact Method for Analysis of Value-based Array Data Dependences. Workshop on Languages and Compilers for Parallel Computing, 1993.
[7] Darte A., Robert Y., Vivien F.: Scheduling and Automatic Parallelization, Birkhaser, 2000.
[8] Lim A. W., Cheong G. I., Lam M. S.: An affine partitioning algorithm to maximize parallelism and minimize communication. In: Proc. of the 13th ACM SIGARCH Int. Conf. on Supercomputing. ACM Press, 1999, pp. 228–237.
[9] www.wolfram.com/mathematica.
[10] Alfred V. Aho, Lam M. S., Sethi R., Ullman J D.: Compilers: Principles, Techniques, and Tools. Addison Wesley, 2006.
[11] Bondhugula U., Hartono A., Ramanujam J., Sadayappan P.: A practical automatic polyhedral parallelizer and locality optimizer. In Conf. on Programming Language Design and Implementation, pp. 101–113, ACM, 2008.
[12] www.nvidia.com

otrzymano / received: 16.07.2010
przyjęto do druku / accepted: 01.09.2010 *artykuł recenzowany*